NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DESIGN OF AN OBJECT ORIENTED AND MODULAR ARCHITECTURE FOR A NAVAL TACTICAL SIMULATOR USING DELTA3D'S GAME MANAGER

by

Rommel Toledo-Ramírez

September 2006

Thesis Advisor:	Perry L. McDowell
Co-Advisor:	Rudolph P. Darken

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2006	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Design of an Object Oriented and Modular Architecture for a Naval Tactical Simulator using Delta3D's Game Manager			**5. FUNDING NUMBERS**
6. AUTHOR(S) Lt Rommel Toledo-Ramirez			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

The author proposes an architecture based on the Dynamic Actor Layer and the Game Manager in Delta3D to create a Networked Virtual Environment which could be used to train Navy Officers in tactics, allowing team training and doctrine rehearsal.

The developed architecture is based on Object Oriented and Modular Design principles; while it explores the flexibility and strength of the Game Manager features in Delta3D game engine.

The implementation of the proposed architecture is planned to be done in standard personal computers running Windows™ OS, but as Delta3D is a multi-platform tool, the generated code can be easily ported to Linux or even Mac™ platforms.

The designed architecture includes also a proposal for fast scenario creation and modification based on XML technology.

14. SUBJECT TERMS Virtual Reality, Virtual Environment, Surface Warfare, Computer Simulation, Computer Graphics, Open source, Division Tactics, Fleet Maneuvers, Surface Tactics, Delta3D, Game Manager, Dynamic Actor Layer, Distributed Simulation.			**15. NUMBER OF PAGES** 75
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

DESIGN OF AN OBJECT ORIENTED AND MODULAR ARCHITECTURE FOR A NAVAL TACTICAL SIMULATOR USING DELTA3D'S GAME MANAGER

Rommel Toledo-Ramirez
Lieutenant, Mexican Navy
B.S. Naval Sciences, Heroica Escuela Naval Militar, 1995

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN
MODELING, VIRTUAL ENVIRONMENTS, AND SIMULATION (MOVES)**

from the

**NAVAL POSTGRADUATE SCHOOL
September 2006**

Author: Rommel Toledo-Ramirez

Approved by: Perry L. McDowell
Thesis Advisor

Rudolph P. Darken
Co-Advisor

Rudolph P. Darken
Chairman, MOVES Academic Committee

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The author proposes an architecture based on the Dynamic Actor Layer and the Game Manager in Delta3D to create a Networked Virtual Environment which could be used to train Navy Officers in tactics, allowing team training and doctrine rehearsal.

The developed architecture is based on Object Oriented and Modular Design principles; while it explores the flexibility and strength of the Game Manager features in Delta3D game engine.

The implementation of the proposed architecture is planned to be done in standard personal computers running Windows™ OS, but as Delta3D is a multi-platform tool, the generated code can be easily ported to Linux or even Mac™ platforms.

The designed architecture includes also a proposal for fast scenario creation and modification based on XML technology.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank Professor Perry L. McDowell for his guidance, patience, advice and support, which were essential for the completion of this research. His work as Executive Director of the Delta3D team is invaluable.

Dr. Rudolph P. Darken: Your guidance in class and during the development of this work, as well as your leadership as Chairman of the MOVES Institute was an extraordinary example to follow.

I am in profound debt to Vice Admiral Alberto Castro, Chief of the General Staff of the Mexican Navy, who believed in me by providing the opportunity for studying this master degree in such an extraordinary Institution. It is an honor to be under your Command Sir.

I also recognize and thank the Delta3D team for their support, especially Chris Osborn, who always had a minute to talk about my project. I also recognize the support of the people in the Delta3D forum for their unconditional help.

To my friend and colleague: Mariano Lizarraga Fernandez for his friendship, support and encouragement.

To the crew on board the Mexican Navy Ship “Tormenta”, for their support during my information gathering travel to their ship.

I gratefully thank my parents from who I always get love, support, and the firm conviction that the most precious of all possessions is knowledge.

Finally, and most importantly, to my family, I thank my son Rommel for been such an inspiration for becoming a better human being every single day, and to my beloved wife Rossy who always had patience, understanding, encouragement and love to me during the whole term on board the Naval Postgraduate School. I dedicate this thesis to you, my love; you know you are my whole World.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND.

The events of September 11, 2001, dramatically changed the whole concept of security worldwide; the Mexican Government and Navy were no exception. The possibility of a terrorist attack upon Mexican assets becomes a distinct possibility.

In 2002, Mexico was the fifth largest oil producer at World level, and the third largest exporter to U.S.[1]. More than three quarters of Mexico’s oil production came from the Cantarell oil field which is the second largest in the World.[2]. The Cantarell oil field is located in the Gulf of Mexico, which is within Mexican control and under the jurisdiction of the Mexican Navy. The production decrease caused by a terrorist attack against the oil rigs in the Gulf of Mexico would damage Mexico’s economy greatly and sharply curtail the supply of oil to the U.S. Additionally, the world economy would suffer greatly from the rise in the per barrel price and.

The one of the most likely threat to the Cantarell field is that a small surface vessel loaded with explosives penetrates the exclusion zone surrounding the oil extraction area and detonates itself. Some previous analysis in this problem reach the conclusion that the speed of the attacking vessels is a strong factor of success for the attack; while on the other hand, early identification of the attacker strongly increase the opportunities of destroying the threat before it can reach its objective.[3]

In order to increase the protection to this valuable national resource, the Mexican Navy acquired from Israel two Aliya Class Missile Patrol Boats, now renamed as “Huracán” Class (Fig. 1) and three E-2C planes with Early Warning System capabilities (Fig. 2). These platforms represent a big step in fleet modernization and increase greatly the capacity of response of the Mexican Navy.

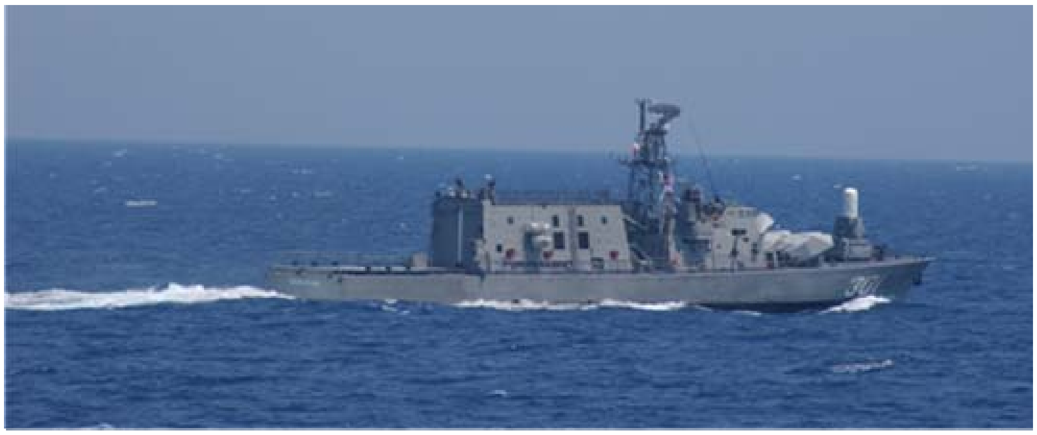

Figure 1. Mexican Huracán Class Missile Patrol

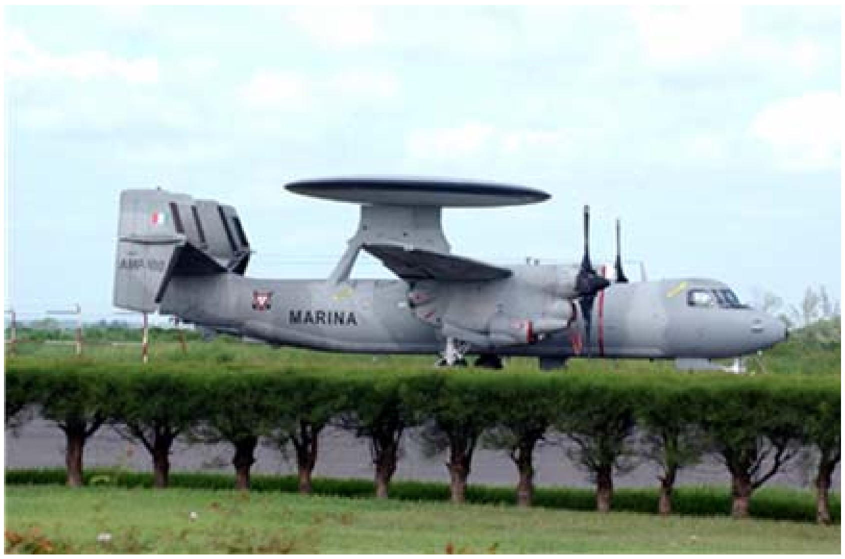

Figure 2. Mexican E-2C Early Warning System Airplane

The addition of these new platforms represents a significant increase in the capabilities of the Mexican Navy and requires new tactics to maximize their effectiveness. The Mexican Navy must come up with a new way to train the officers and crews of these platforms in these tactics.

Currently, the Mexican Navy trains tactics to the Officers in the CESNAV[1], where theoretical concepts are taught in the classroom, and then students have

[1] Spanish Acronym for: Centro de Estudios Superiores Navales. (Naval Center for Superior Studies)

the opportunity to get some insights from the “Wargame”, which is conducted using a strategic naval warfare simulator bought in the late 90’s from a U.S. contractor. However, the Navy does not possess a tactical trainer in which Mexican naval officers can gain insights of their classes at a tactical level.

Because the Mexican Navy has upgraded its inventory, it creates the necessity of improving the training of the personnel aboard the new ships and the officers who will command them.

B. PROBLEM STATEMENT

This thesis work is a research in the design of architecture for a Networked Virtual Environment (NVE) to training and/or rehearsing the procedures aboard a Surface Unit, specifically modeled in here the “Huracán” Class Ships, in the event of Anti-Surface Warfare (ASuW). The designed architecture is based on the current development of the Delta3D game engine.

This work addresses the following questions:

The feasibility to build a Naval Surface warfare training environment in a virtual world, where models for sensors, movement of units, weapons performance, etc., can be added without modification of the main game engine.

The possibility of building such a NVE using the Delta3D Game Engine. If so, the viability to build a Modular Architecture Design using the Dynamic Actor Layer and the Game Manager features in Delta3D.

Developing an architecture that allows the use of public data for units, sensors, and weapons, in developing time, and classified data in deployment time.

The design that allows easy-creation scenarios in the described NVE

C. THESIS OUTLINE

Following this introduction, Chapter II describes the objectives and limitations relevant to the designed architecture. Chapter III makes a brief explanation of the Delta3D game engine and some of its components, such as the Game Manager and Dynamic Layer architectures. Chapter IV describes the core of the proposed architecture and the principles used to develop it. Chapters V through IX describe each component part of the architecture, Chapter X covers the integration of all constituent parts, and Chapter XI presents the conclusions and proposes future work to improve this thesis.

II. CHARACTERISTICS AND LIMITATIONS OF THE ARCHITECTURE DESIGN

A. DEVELOPMENT ANTECEDENTS

During the modernization effort started in recent years, the Mexican Navy created the INIDETAM[2], a research institute to investigate how best to employ the Navy's weapons systems. The INIDETAM is currently conducting research to develop a tactical simulator for the Mexican Navy. This new simulator has to allow students to interact in a CIC[3] environment, employing computer software to simulate tactical theaters for Naval Warfare.

This simulator will be used in the CESNAV as a teaching aid in the theory of Naval Tactics to officers, as well as experimentation of new tactics in the operational theater.

The main objectives for this simulator are to provide a training tool for:

- The application of tactics for Navy Officers.
- The officers in the tactical use of sensors, electronic warfare equipment and weapons systems.
- The use of naval units in the development of tactical exercises (maneuvering and formations)
- Tactical decision making.
- Target Motion Analysis (TMA).

B. GENERAL DESCRIPTION

The objective of this thesis work is to design an architecture to develop a Networked Virtual Environment (NVE) and create a test bed application using

2 Spanish Acronym for: "Instituto de Investigación y Desarrollo Tecnológico de la Armada de México" (Mexican Navy Research and Technology Development Institute).

3 Combat Information Center.

this architecture. The resulting products could be used to support the further development of the Naval Tactical Simulator for the Mexican Navy.

Networked Virtual Environments are tools that allow multiple users to interact in real-time in a shared virtual world. Modern NVE's can represent the shared world in realistic 3D graphics and sounds, giving the feeling of realism and/or immersion and providing a better experience to the users. This feeling of realism and immersion can be used to train personnel in some tasks that for its own nature are too expensive or risky to perform in reality.

The proposed architecture will be implemented in a test bed application, simulating different watch stations that take part in the process of ASuW onboard a surface unit.

This development provide a model for the design and implementation in a graphical interface with network capabilities while it is flexible and modular enough to allow further inclusion of sensors, weapons and units models without extreme changes to the main architecture.

C. MAIN CHARACTERISTICS

The development of the architecture is based on the current development stage of the Delta3D game engine.

In order to achieve the benefits of team training a multiplayer architecture is used in the architecture.

The developed architecture and test bed application simulates the bridge and the different ASuW consoles of the CIC aboard a "Huracán" Ship. In order to create an application within the time constraints of this thesis, some consoles functions were simplified. The consoles were changed in layout and/or content to keep this work unclassified.

The proposed architecture has the following characteristics:

- PC technology based.

- Windows platform based.[4]
- Client-Server Network configuration.
- Modular Design.
- Object Oriented.
- Implemented in the Delta3D game engine using the Game Manager and Dynamic Actor Layer capabilities.

D. MAIN LIMITATIONS

The author planned to design this architecture in such a way that it allows its growth in the future to other kinds of naval warfare (ASW, AASW, etc.), without substantial changes to the design. The generated application is intended as a proof of concept and a test bed for the designed architecture; it is not a finished product that can be used immediately for training.

While the creation of physics models for sensors, weapons, and units is out of the scope of the research, some models were created for testing purposes. The game artificial intelligence necessary to control opposition forces, friendly forces and neutral units are also out of the scope of this thesis. Other desirable characteristics, for instance recording and playback functions for after action review, were not considered, even though the Game Manager tool inside Delta3D will make further implementation significantly easier to incorporate.

E. REASONS TO USE THE DELTA3D GAME ENGINE

The Delta3D game engine was chosen because it is supported and funded by the MOVES Institute and the U.S. Department of Defense (among others). This engine is completely capable for the development of a full-size training simulation application.

4 It is possible to develop the application program in Linux or other compatible OS supported by Delta3D.

By using the Delta3D open source game engine, a surface warfare trainer can be built that is capable of teaching new and experienced Mexican naval officers tactical training that they could only otherwise achieve in costly and/or hazardous naval exercises. The open source trainer could be used for low level learning such as teaching the fundamentals of what is learned at each watch station, up to high level tactical thought processes of the senior officer in charge of each ship. The trainer would utilize existing personal computer hardware and software, and would be scalable to include single ship operations through whole fleet interactions. The open source nature of the architecture would allow for further improvements to the simulation by the Mexican Navy. With the source code available, changes in equipment or doctrine could easily be changed in the engine to allow it to stay current.

Because the Delta3D engine is licensed under the Lesser Gnu Public License (LGPL), future developments based on the proposed architecture and companion application to be restricted for the use of the Mexican Navy. This will allow the Mexican Navy to create and maintain its own technology in the field of simulation-based training.

Finally all the knowledge and skill of the Delta3D team was available on site to support the author in the development of this thesis work.

III. DELTA3D'S GAME MANAGER AND DYNAMIC ACTOR LAYER OVERVIEW

A. WHAT IS THE DELTA3D ENGINE?

As described above, the Delta3D is a game engine which is in constant development in the Modeling, Virtual Environments and Simulation Institute (MOVES).

According to the Wikipedia, a game engine is:

> …the core software component of a computer or video game or other interactive application with real-time graphics. It provides the underlying technologies, simplifies development, and often enables the game to run on multiple platforms such as game consoles and Microsoft Windows. The core functionality typically provided by a game engine includes a rendering engine ("renderer") for 2D or 3D graphics, a physics engine or collision detection, sound, scripting, animation, artificial intelligence, networking, and a scene graph [4].

The Delta3D game engine is a design that encloses other well-know Open Source libraries as Open Scene Graph, Open Dynamics Engine, Character Animation Library, and OpenAL as the underlying modules of the engine. It provides scripting, physics, animation, special effects, terrain, networking and sound capabilities; and it can be used under Windows, Linux and recently Mac operating systems.

In version 1.0, Delta3D introduces the concept of Dynamic Actor Layer, and in its 1.2 version the concept of Game Manager. Both are in a fully stable stage in the current version (as of September 2006), version 1.3.

B. WHAT IS THE DYNAMIC ACTOR LAYER?

In order to be capable of managing all kinds of applications in Delta3D from military simulations to leisure games, the concept of Dynamic Actor Layer (DAL) was created.

Using this concept, the game engine can manipulate all kind of user created actors by exposing them to the engine as a generic template that it can understand. Using this concept a rock is not different from a naval carrier vessel.

> The core of this concept is that each actor should be treated generically using only its name, id, and a collection of properties.[5]

The DAL component is composed by three different parts:

- Library Manager
- Actor Proxy
- Project

The Library Manager is responsible for knowing about what actors libraries are being used inside the simulation, and it has mechanism for finding actors and creating them dynamically.

Actor Proxy is a wrapper for different actors; actors are user created pieces of code that implement different entities inside the simulation, from trees and buildings to vehicles and weapons. The actor proxy translates the actor class and exposes its properties using simple “Getter” and “Setter” methods.

Project provides a common file structure for resources inside libraries, and it also avoids having duplicated resources and facilitates distribution of the final application.

C. WHAT IS THE GAME MANAGER?

In order to increase software usability and speed development the Game Manager Architecture was created inside the Delta3D engine,

> It is responsible for managing both game and non-game actors, ensuring messaging and inter-object communication, and directing component behavior. It knows which components exist, what actors are interested in which messages, and what actors exist in the simulation. It is then responsible for making sure messages get to the components and interested actors as well as handling low-level events from the Delta3D system such as frame and pre-frame events.[6]

Game Manager is composed on three main parts:

- Messaging Architecture.
- Game Actors Proxies and Game Actors
- Components.

Messaging Architecture provides communications between the different actors that live inside the simulation. These messages go to and from every object inside the simulation (actors and components). Messages are the way in which game events are communicated in the simulation. Since every message if passed to the Game Manager, it can forward them to a remote game server or perform validation.

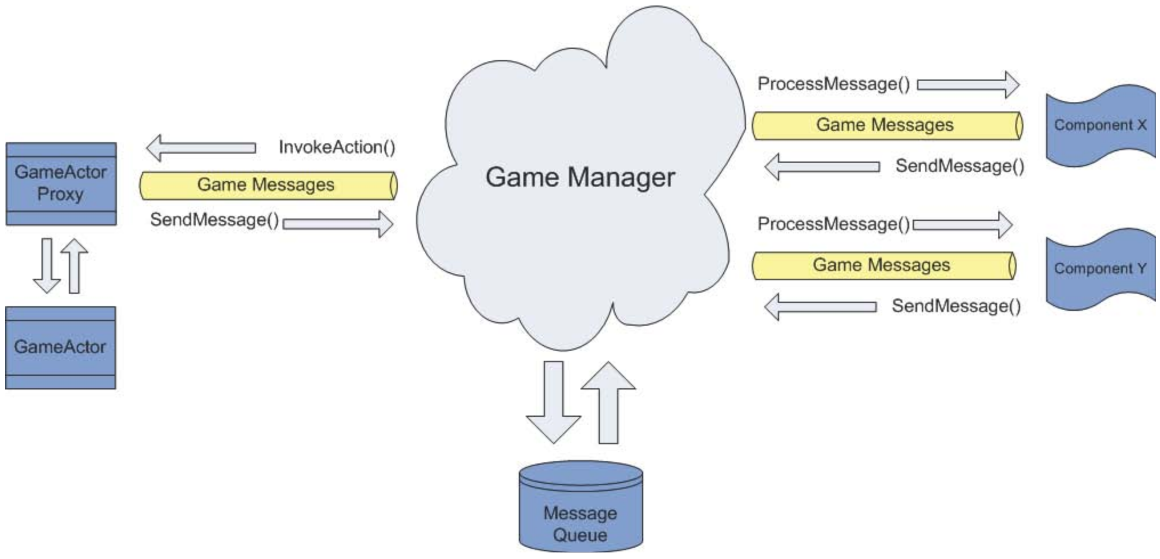

Figure 3. High Level View of Message Traffic (From Ref. [6])

Game Actor Proxies and Game Actors are similar to regular Actors and Proxies as decribed above, but they inclued a minimum set of properties, for instance location, scale and rotation. Additionally, they also include invokable methods, they understand the difference between local and remote and can fire different behaviors needed by a Game Manager such as entering the world or updating their own data.

Components are special kind of objects inside the Game Manager that receive every message in the simulation and can handled them in a higher level than Actors, components can perform different sets of task, from networking to rules validation; components can also be added to the simulation dynamically on runtime or compile-time.

IV. ARCHITECTURE DESIGN OVERVIEW

A. THE PROPOSED ARCHITECTURE

The author propose an architecture based on the Dynamic Actor Layer and the Game Manager in Delta3D to create a Networked Virtual Environment which could be used to train Navy Officers in tactics, allowing team training and doctrine rehearsal.

The developed architecture is based on Object Oriented and Modular Design principles; while it explores the flexibility and strength of the Game Manager features in the Delta3D game engine.

The implementation of the proposed architecture is planned to be done in standard personal computers running Windows™ OS, but as Delta3D is a multi-platform tool, the generated code can be easily ported to Linux or even Mac™ platforms.[5]

The designed architecture includes also a proposal for fast scenario creation and modification based on XML technology.

B. MODULAR DESIGN OF SIMULATION ENTITIES

> Modular design means trying to subdivide an assembly in smaller parts (modules) that are easily interchangeably used.[7]

This architecture handles simulation entities as a sum of elements. These elements are provided in a very generic way and can be individualized from data obtained from the scenario file. This is useful because this data can be taken from open or public sources for the development phase and can be easily changed to classified sources when the system is finally complete and ready to be used as a trainer or a rehearsal tool.

The simulation entities mentioned above could be ships, submarines, aircrafts, weapons or facilities.

[5] The current version of Delta3D (1.3) supports the following operating systems: Windows™, Linux and MacOSX™.

The elements that constitute the simulation entities are: sensors, communications devices and weapons systems. Some elements could be a sum of elements too.

For instance, the Mexican Huracán Class Missile Patrol shown in the following figure (Fig. 4) is built inside de simulation using the following elements:

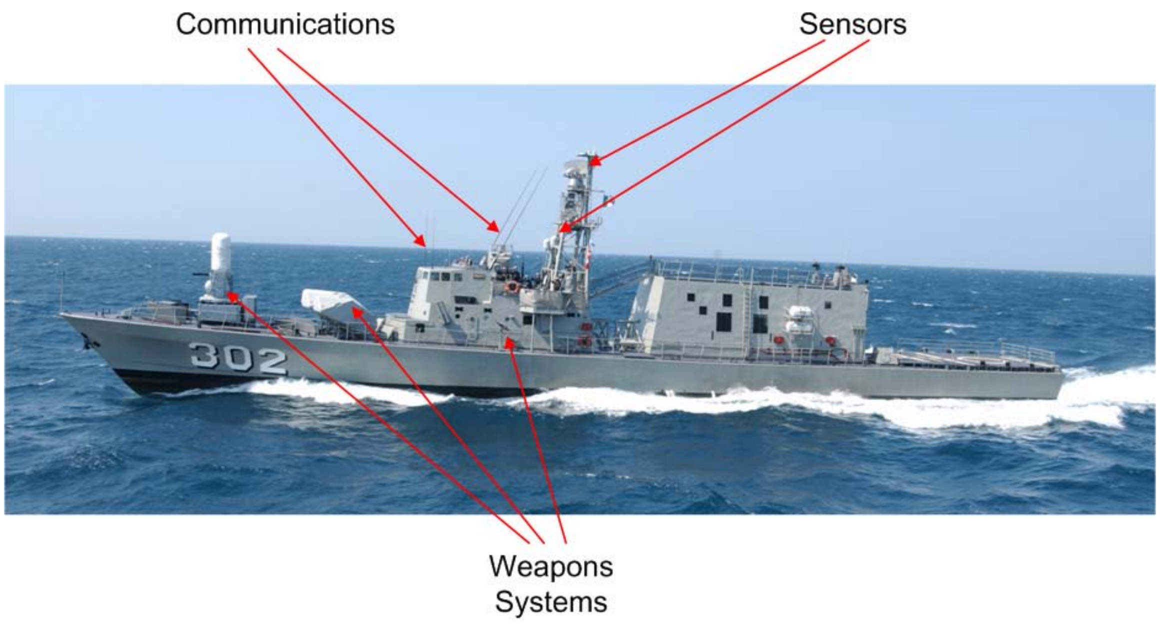

Figure 4. Mexican Huracán Class Missile Patrol showing simulation elements

- Communications:
    - Gabriel Command Data link
    - Tadiran UHF
    - Tadiran VHF
- Weapons Systems
    - 12.5 mm/50 Machines Guns
    - 20 mm Oerlikon Machines Guns.
    - Gabriel II Anti-ship Missile
    - Mk 15 Block 0 CIWS System.

- Sensors
    - Neptune/ TDH 1040 Radar
    - Galileo OG 20 Optical Director
    - NS 9003 ESM
    - NS 9005 ECM

Also weapons systems can be created by summing up elements, for instance the Mk 15 Block 0 CIWS System on board the Huracán Class is built by these parts shown below (Fig.5):

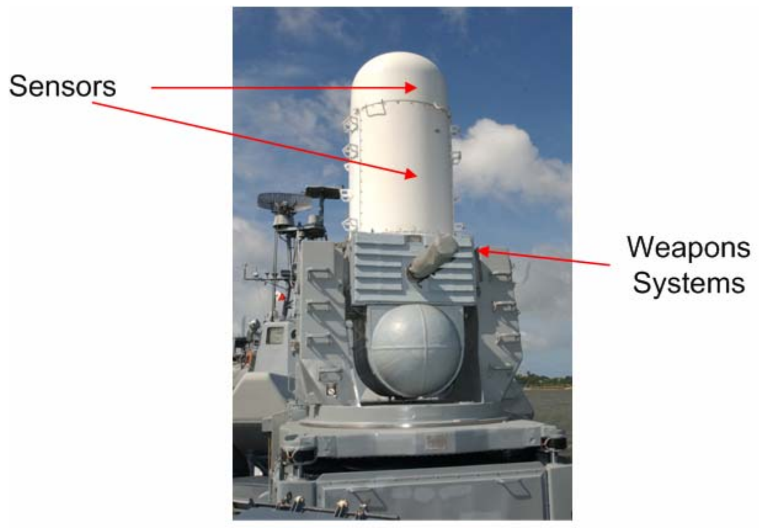

Figure 5. Mk 15 Block 0 CIWS showing simulation elements

- Sensors
    - Search Radar
    - Tracking Radar
- Weapons
    - M61 Vulcan Gatling gun

Even single pieces of ordnance, such as missiles, can be represented as a sum of elements, for example de Gabriel II Missile carried by the Huracán Class can be described by the parts in the figure below (Fig. 6):

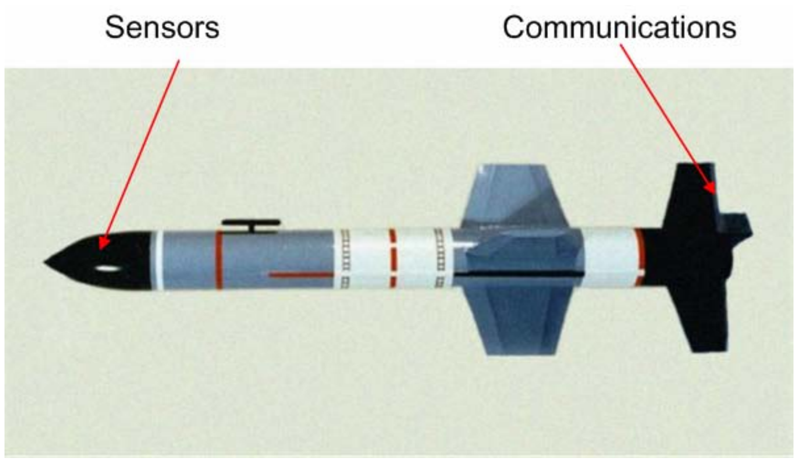

Figure 6. Gabriel II Missile showing simulation elements

- Sensor
  - Semi active homing radar
- Communications
  - Gabriel Data Link System

The actual ship data as length, displacement, crew count, and cruise speed, etc., or CIWS data as rate of fire, range, etc., or missile data as top speed, flight height, etc., is also obtained from the scenario file.

Given that there exist containers for all kinds of elements in the simulation, different kind of units (ships, submarines, aircraft, weapons or facilities) can be created in runtime without changes in the simulation code, simply by choosing different elements to build a new simulation entity in the scenario file.

The behaviors required for controlling these units are created using the same approach. Different modules for physics or artificial intelligence are designed as separated entities giving more flexibility to the units that can be created inside this simulation. Chapter V describes these concepts in detail.

C. DESCRIPTION OF THE OBJECT ORIENTED DESIGN

> The idea behind object-oriented programming is that a computer program may be seen as comprising a collection of individual units, or *objects*, that act on each other, as opposed to a traditional view in which a program may be seen as a collection of functions, or simply as a list of instructions to the computer. Each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent little machine or actor with a distinct role or responsibility.[8]

As described in the previous section of this chapter, every single simulation entity is based on a sum of elements; these elements are to be coded as different classes inside the simulation.

To increase software reusability and modular design, every class which represents those entities is designed to store data sufficient to create an instance of that given entity, plus containers to stores references to each conforming element. Using this approach, all information required to build a single unit is self contained in one single object, providing better encapsulation. This concept is cover in detail in Chapter V.

D. ARCHITECTURE DESIGN COMPONENTS

The described architecture is dived in five main constituents:

- Game Actor Library
- Scenario Creation, Edition and Configuration
- Network Structure
- Realistic Console Displays Representation
- Custom Game Manager Components

These constituents are described in following chapters of this thesis work.

THIS PAGE INTENTIONALLY LEFT BLANK

V. GAME ACTOR LIBRARY DESIGN

A. GAME ACTOR LIBRARY OVERVIEW

Actors are the fundamental building pieces of a simulation; they perform all different kind of activities, behaviors and tasks. The Game Manager Architecture provides a general structure that allows the creation of a wide variety of actors, from simple to very complex ones.

The Game Actor Library designed for the proposed architecture provides the building blocks for most common entities inside a naval warfare simulation; these actors are all inherent from the GameManagerActor class of the Delta3D API[6]. It is represented as a tree of C++ classes, as shown in the following figure (Fig. 7):

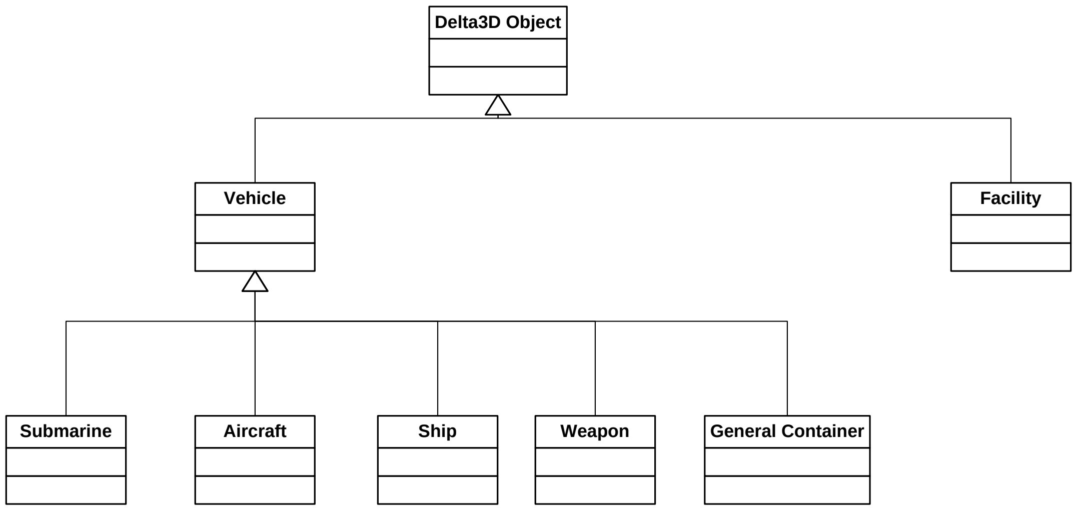

Figure 7. High Level View of Game Actor Library

There exist two main kinds of actors inside this design: Vehicles and Facilities.

Vehicles are any kind of actor that is capable of movement inside the simulation, as ships, submarines, aircraft, etc.

[6] Application programming interface.

Facilities are any kind of actor that is geographically fixed in the simulation, as oil rigs, military bases, airports, etc.

The Game Actor Library is a group of actors which represent all kind of possible generic platforms inside the simulation. This group of actors is an implementation of the Game Manager Layer architecture of the Delta3D engine.

In this thesis work a Building Element refers to sensors, communication devices or weapons as they constitute the build blocks of any platform inside the simulation, as described in Chapter IV.

B. ACTOR DESCRIPTION

Given that the Game Manager architecture already provides some generic data for actors, such as: location, scale, and rotation, the Vehicle Class shown in Figure 7 was not implemented for not being necessary, and moveable actors are directly inherent from the GameManagerActor class.

Basically, Facility Actors do not use information about position in other time than when they are created, so they can located inside the simulation and they are not updated in position.

As shown in Figure 7, Vehicles Actors are subdivided into more classes; these classes are used to represent different platforms types inside the simulation.

Every actor in the simulation works as a container for data and for the Building Elements. The type of data that is stored in each actor depends on the kind of actor that is being created, and it is loaded from the scenario file when the actor is created, the number and type of sensors, communication devices and weapons system is given by the scenario file too. Every actor can own from zero to n of these Building Elements.

The General Container Class shown in Figure 7 represents a special case of actor that is able to represent any object not already designed in this architecture, it can be positioned in any place inside the simulation and its also capable of movement, allows the use of Sensors, Weapons or Communications devices. For instance, it can be used to represent a sonobuoy, which is basically a sensor with a communication devices for linking purposes.

The following is a detailed description of the Ship Actor. Given that all kinds of Actors are constructed in the same way and only particular data changes inside them, this can be seen as an example of how all actor classes are constructed.

C. THE SHIP CLASS

As have been stated before, every actor class works as a container of data and Building Elements given by the scenario file.

The following Entity-Relationship diagram in Figure 8, defines some of the data contained in the Ship Actor class. The data shown is for testing purposes only while this architecture is designed and tested, this data can be extended to store different or additional information in further versions.

This diagram represent the different data that is used inside a entity in the simulation, in this case, platforms and Building Blocks are those entities, the relationships between the data in the entities and the relationship between the different classes that represent those entities inside the simulation could be different from what it is being represented in the following diagram.

The Ship Class is related to the Weapons, Sensors and Communications classes, which contain their own data as well. The relationship between them is a many-to-many relationship, which means that the same sensor type can be used several times in the same ship, or the same sensor type can be used several times in different ships (or other types of units).

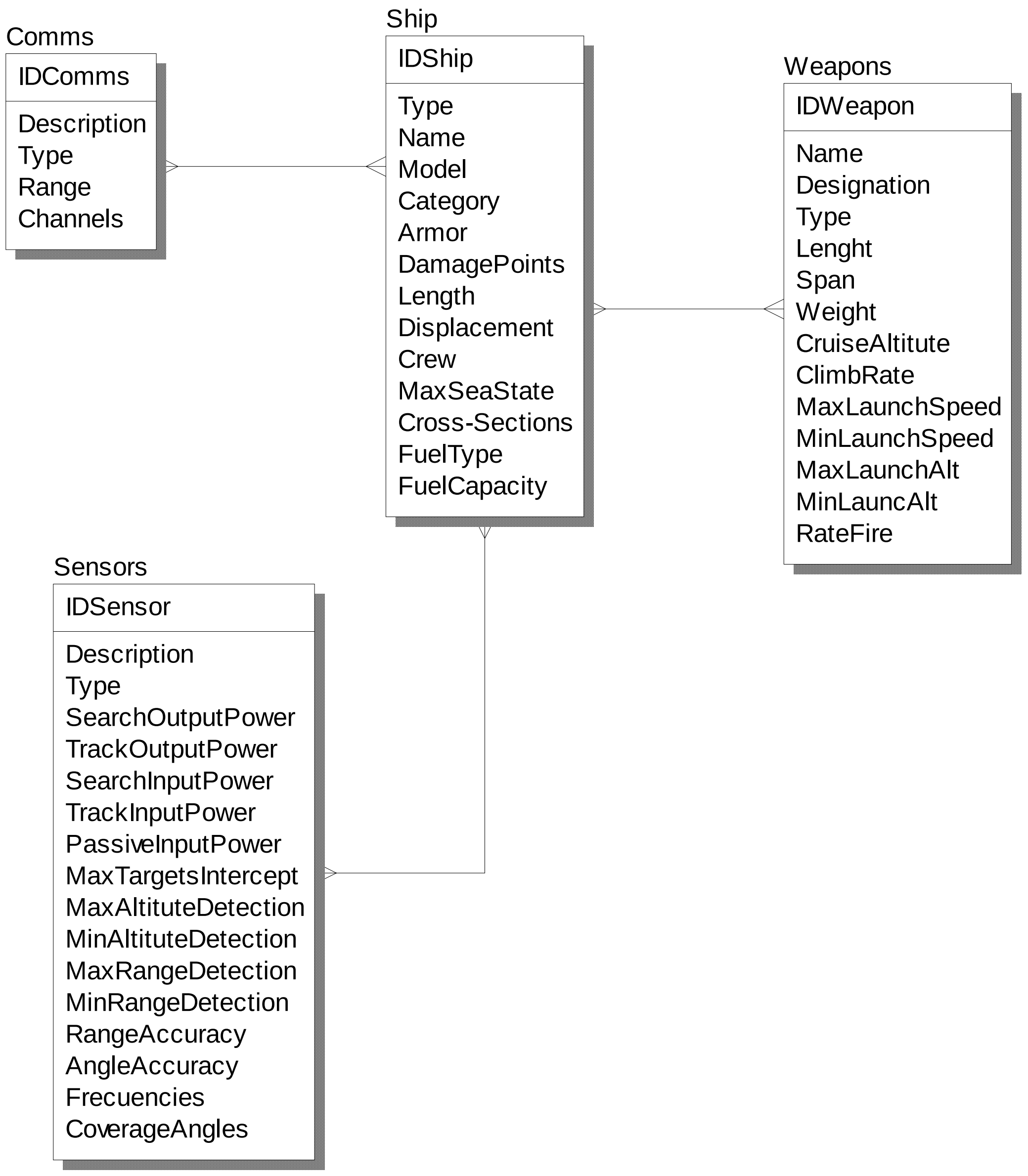

Figure 8. Relationship between the Ship Actor Class and the Building Elements Classes.

The ship class (and all other actor classes) has some general methods inherited from the Game Manager Actor class; the following figure (Fig. 9) shows some of them as well as some custom methods required to the class function.

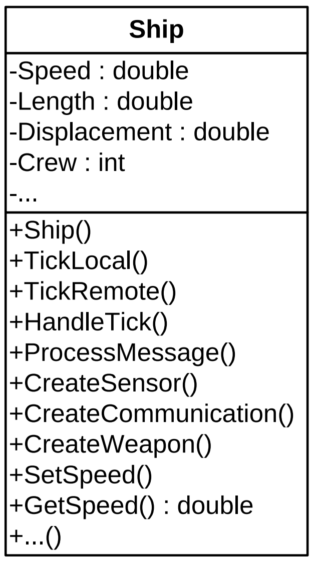

Figure 9. Ship Classes showing attributes and Methods

The TickLocal() and TickRemote() methods will respond to TickMessage from the Game Manager Architecture; both methods will call the method HandleTick() which will call all methods required to be performed per frame. For instance, these include sensors' detection algorithm or position update given the current speed and heading.

The CreateSensor() method is used when the Ship Actor is created to create a new sensor object and assign it to the Ship instance. The same applies for the CreateWeapon() and CreateCommunications() methods.

The method ProcessMessage() is inherited from the Game Manager Actor class and it is used to handle all simulation events such as: increase speed, turn, etc. Game events will be covered in more detail in Chapter IX.

The Dynamic Actor Layer required that all attributes in the class has its corresponding Setters and Getters methods which are also provided in this implementation.

The Actor classes have platform specific methods (not shown in the previous Figure); such methods are called in response to a Game Event from the

ProcessMessage() method or can be auxiliary methods to other ones, example of this are StartShipEngines(), FlankSpeed(), etc.

D. SENSORS, WEAPONS AND COMMUNICATIONS CLASSES

In order to keep the implementation of the different constituent parts of the platforms separated from the platform itself, Building Elements are designed in different classes by creating instances of the Weapons, Sensors and Communications Classes.

The Sensors and Communications classes are not inherited from the GameManagerActor class, but they are closely related to the classes which are in fact inherited from it, and they are in fact stored inside the same Dynamic Library which stores the Actors Classes.

The Weapons Class is in fact two different classes. One of them is an implementation of the GameManagerActor class and is inherited from it. This is because some kind of weapons, such as missiles and bombs, can be drawn inside the simulation (they required to be represented with a 3D mesh). Thus, it must possess all methods described above for Actor Classes, as well as its own specific methods. When a weapon is non-drawable, like artillery ammunition, the weapon is created as a non-actor class and is not capable of receiving Game Events from the simulation and is neither represented as a 3D model inside the application.

The Sensor Class, for example, provides with all the data storage capability to create any type of sensor (Radar, Sonar, Optical, ESM, etc.). When it is created, it loads all required data from the scenario file and based on the kind of sensor it represents and it chooses the proper detection algorithms from the corresponding supporting class described later in this chapter.

In this way, generic classes are used to represent any kind of sensor, which reduces the number of classes that need to be designed and leaves individualization of the sensor to the scenario file. The same applies for the Weapons and Communications classes.

E. MODULAR SUPPORTING CLASSES

Beside the Building Elements and data required to construct a Ship Actor, some other classes are required to control the actor or provide certain capabilities. Artificial intelligence and physics are two examples of this.

All capabilities required for the proper Actor operations are code in separated classes, which allows future enhancement without modifying the main Actor Class.

The following picture (Fig. 10) shows how some example classes are related to an Actor Class and to a Building Element Class.

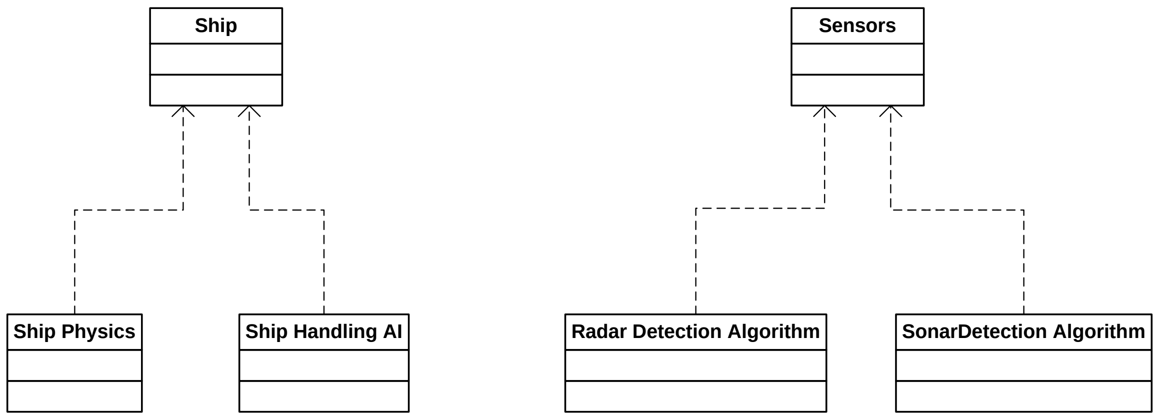

Figure 10. Relationship between Actor Classes and supporting classes

The previous Figure shows how the Actor Ship Class makes use of the Ship Physics and ShipHandling AI Classes, while the Sensor Class makes use of the RadarDetectionAlgorithm and SonarDetectionAlgorithm classes. In both cases, the supporting classes make use of standardized method calls. For instance, if the Sensor Class makes a call for IsTargetInRange() method, either Radar or Sonar detection algorithms will be applied depending on the specific sensor type that perform the method call.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. SCENARIO CREATION, EDITION AND CONFIGURATION

A. SCENARIO DESCRIPTION AND FILE CONFIGURATION

Scenarios provide the prime configuration and the starting population of actors in a simulation. They also provide actors' positions, weather configuration and the virtual spatial location of the entities being simulated. This architecture makes use of XML to store scenario information in a file which is loaded to populate and configure the simulation. This scenario file can be created manually or by a database oriented implementation which will be described later in this chapter.

One description of XML states that:

> The Extensible Markup Language (XML) is a W3C-recommended general-purpose markup language for creating special-purpose markup languages, capable of describing many different kinds of data. In other words, XML is a way of describing data. An XML file can contain the data too, as in a database. [9]

These capabilities are used to store in a XML file all information related to platforms and their Building Elements, as well as the weather configuration that is implemented using the weather rendering capabilities of the Delta3D engine. Delta3D provides a XML parser that can be customized to read specific XML files and/or schemas. Additionally, the game engine provides terrain rendering, which is out of the scope of this thesis, but is included as a possibility for a future enhancement to the architecture to allow littoral operations.

The scenario file used by this architecture is divided into three main areas:

- Scenario and Weather Data
- Actors' Data
- Terrain Data

1. Scenario and Weather Data

This section of the scenario file stores all information relative to the scenario itself, this information is subdivide in several subsections:

- Scenario Information
- Area Information
- Weather Information

A fragment of the Scenario file is shown in the following figure (Fig. 11).

```
<Scenario xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <ScenarioID>001</ScenarioID>
        <ScenarioName>Sonda de Campeche</ScenarioName>
        <ScenarioDate>03/31/06</ScenarioDate>
        <ScenarioTime>12:38</ScenarioTime>
        <SimDate>25/05/2006</SimDate>
        <SimTime>08:43</SimTime>
        <SimArea>
                <UpperRightCorner>
                        <Latitude>23.5N</Latitude>
                        <Longitude>124.4W</Longitude>
                </UpperRightCorner>
                <LowerLeftCorner>
                        <Latitude>20.5N</Latitude>
                        <Longitude>134.4W</Longitude>
                </LowerLeftCorner>
        </SimArea>
        <WeatherConditions>
                <Clouds>3</Clouds>
                <Rain>0</Rain>
                <SeaState>3</SeaState>
                <Fog>3</Fog>
                <WindForce>2</WindForce>
                <SeaCurrent>
                        <Speed>3</Speed>
                        <Course>310</Course>
                </SeaCurrent>
        </WeatherConditions>
```

Figure 11. Scenario Information and Configuration Section (with data)

The Scenario Information section provides the data required as information for users about the file itself. The information about the date and time being simulated is also contained in here.

Area Information provides the virtual geographical area where the simulation takes place.

Finally Weather Information provides the Game Manager Server with the information required to create the proper weather representation. Currently, Delta3D supports sky dome, fog and wind speed which is used to determinate the speed of the clouds in the sky. These numeric values, as well as all others shown in the previous figure, are globally available for the actors and Custom Game Components which can use them to perform several calculations that are weather-depend, such as: radar detection range, units speed, visibility, etc.

2. Actors' Data

As described previously, actors' information is stored in the Scenario File, the actors section is subdivided into two main subsections which are divided further to store all information relative to the Building Elements of every Actor.

The Actors Section is divided in:

- Vehicles
- Facilities

The Vehicles subsection stores all information relatives to actors capable of movement, which are Ships, Aircraft, Submarines and General Containers.

Chapter V states that there exists a Weapon actor which is cataloged as a mobile actor for the scenario file purpose. The Weapon Actor is considered to be a building part of platforms, so it is described in the weapon subsection of each platform as will be shown later in this chapter. Given the fact that there exist two different kinds of Weapon Actors, drawable or non-drawable, the Scenario File provides the information relative to the Weapon Actor type, so it can be correctly created in the simulation when needed.

The facilities subsection is not subdivided, given that every facility is considered to be equally functional as any other, and there is not information to differentiate between them.

The Vehicles Subsection is subdivided in four sub-subsections as follows:

- Ships
- Aircraft
- Submarines
- General Containers

The structure used to represent the described information in the XML file is shown in the following figure (Fig. 12).

```
<Vehicle>
        <IDVehicle>Blue01</IDVehicle>
        <Side>Blue</Side>
        <X>0.0</X>
        <Y>0.0</Y>
        <Z>-1.0</Z>
        <Pitch>0</Pitch>
        <Roll>0</Roll>
        <Yaw>0</Yaw>
        <ScaleX>2</ScaleX>
        <ScaleY>2</ScaleY>
        <ScaleZ>2</ScaleZ>
        <Speed>0</Speed>
        <Course>0.00</Course>
        <BattleGroup>None</BattleGroup>
        <Mesh>..\\Data\\Models\\tormenta.3ds</Mesh>
        <PlataformType>
                <Ship/>
                <Aircraft/>
                <Submarine/>
                <GeneralContainer/>
        </Vehicle>
```

Figure 12. Vehicle Subsection of the Scenario File showing general vehicle data.

All four sub-subsections contain information that is common for all of them. This information contained is provided to located the actor in the simulation, rotate it, scale it, provide starting speed and course, battle group to which the unit belongs and 3D model file name. The scale data is particularly important because 3D models from different sources can be used in the simulation. When these models are not scale coherent, they can be easily be fixed to represent scale coherent units just by changing the value in this subsection without the need to fix them using 3D-modeling tools.

Following the general vehicle information is the information relative to each platform type. For description purposes, the Ship Actor data structure will be described in this chapter; however, all other platforms follow the same structure and use their own required data.

a. An Actor and its Building Elements Data Structure

As described in Chapter V, there exist several actors classes, and the information contained in the scenario file required for building such actors is stored in a structure that emulates the class data. In the following figure (Fig. 13) the Ship data structure is shown as it is stored in the XML Scenario file.

Every piece of data is used to create a Ship Actor Class object in the simulation. As described previously every platform is conformed of several Building Elements, the data structure also includes subsections to store all required data to construct this Building Elements.

As observed in Figure 13, the data structure inside the Ship Actor contains Sensors, Communications and Mounts subsections, all of them can contains from zero to n Building Elements definitions.

```xml
<Ship>
    <IDShip>ARM-02</IDShip>
    <Type>Missile Patrol</Type>
    <Name>Tormenta</Name>
    <Class>Huracan</Class>
    <Category>Navy</Category>
    <Armor>None</Armor>
    <DamagePoints>45</DamagePoints>
    <Length>61.7</Length>
    <Displacement>488</Displacement>
    <Crew>53</Crew>
    <MaxSeaState>5</MaxSeaState>
    <RCSRadar>Small</RCSRadar>
    <RCSVisual>Medium</RCSVisual>
    <RCSIR>Small</RCSIR>
    <RCSSonar>Noisy</RCSSonar>
    <AirFacilityType>Small HeloPad</AirFacilityType>
    <FuelType>Diesel</FuelType>
    <FuelCapacity>3737.63</FuelCapacity>
    <Speeds>
        <Throttle1>0</Throttle1>
    </Speeds>
    <Notes>None</Notes>
    <Sensors>
        <Sensor/>
    </Sensors>
    <Communications>
        <Communication/>
    </Communications>
    <Mounts>
        <Mount>
            <Weapons>
                <Weapon>
                    <Sensors/>
                    <Communications/>
                </Weapon>
            </Weapons>
            <Sensors/>
            <Communications/>
        </Mount>
    </Mounts>
</Ship>
```

Figure 13. Ship data structure in the Scenario File showing data for example purposes.

The Mounts subsection recursively contains Sensors and Communications subsections, as well as the Weapons subsection. This again recursively contains Sensors and Communications subsections. All this information is provided in order to construct the actors in a modular basis as described in Chapter V.

Facilities are structured in the same approach as Vehicles, with the only difference that Sensors, Communications and Mounts subsections are directly attached to the Facility data instead of being dependant of a Platform Type subsection.

The actor information will also contain the indication of the expected type of player that will manage it in the simulation: human player or computer.

3. Terrain Data

While terrain representation is out of the scope of this thesis, the Delta3D game engine allows in its more current version several ways to represent and load dynamically generated terrain, such methods are:

- SOARX
- DTED
- LCC
- GEOTIFF

The terrain representation is considered a future enhancement for this architecture.

B. DATABASE-BASED SCENARIO CREATION

In order to store all information required to build the actors described in previous sections and chapters of this thesis, a database based approach is proposed.

All data required to build a Sensor, Communication Device or Weapon will be stored in a relational database, and then the user can create new units (ships, aircrafts, submarines, etc.) by inputting the required data for the desired platform type and attaching the data from the Building Elements already stored in the database. Finally the new unit will be stored itself in the database for future use. A representation of this architecture is shown in the following figure (Fig 14).

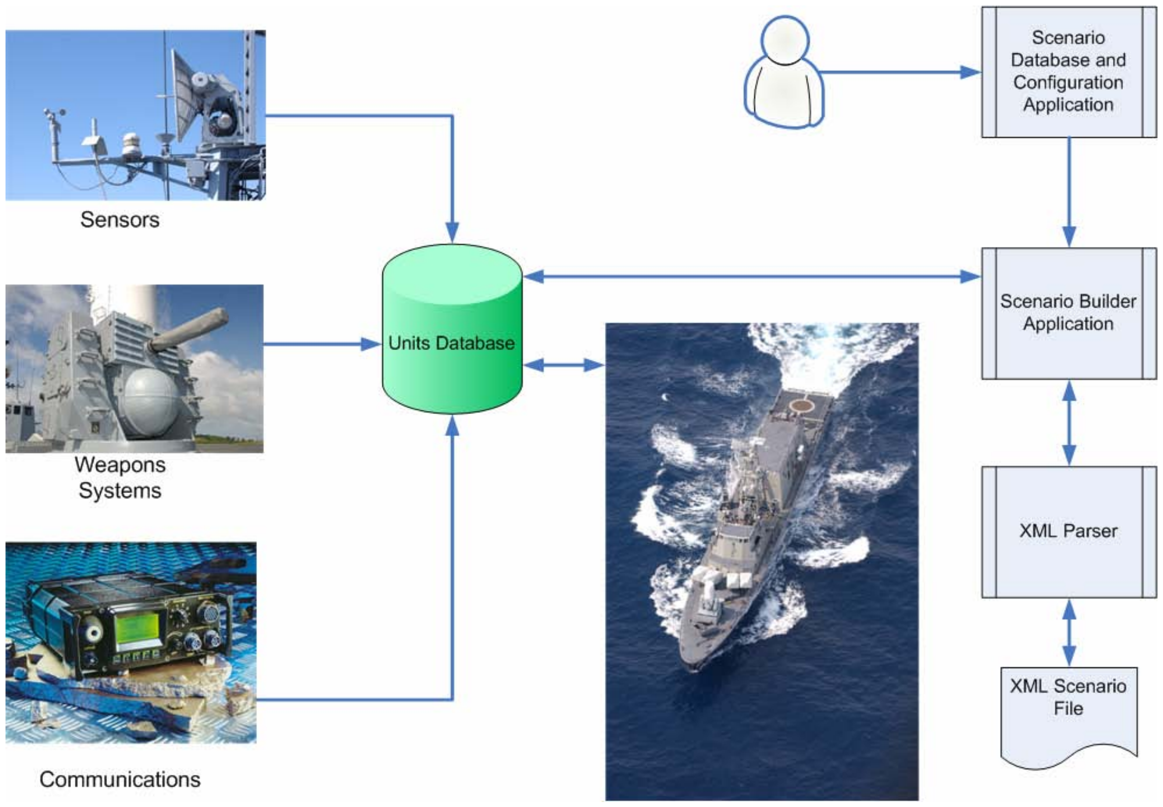

Figure 14. Block diagram of the database-based scenario creation architecture

To build a scenario, the user will input all scenario configuration information as described in the previous section of this chapter (Scenario and Weather data) and assigning as many as necessary or desired units of any type that are required, while assigning the different actors the user can located them, by providing the required information. When the scenario configuration is complete, the user will export all the data using a XML parser which will output the Scenario File with the proper structure described in this chapter.

This approach allows the Scenario Builder Application to read from a previously created XML Scenario file to edit it or save it as other name to create a new one based on a previous edition. The proposed configuration is very flexible, so new units can be created using any combination and number of Building Elements and providing the required data to represent them in the simulation.

Finally as weapons are considered a composite Building Element, they can be built and stored using the same approach described for units.

C. CONFIGURATION OF THE SIMULATION SCENE FROM THE XML SCENARIO CREATION FILE

Following a process similar to the creation, but in inverse direction, a XML scenario file can be read to populate the simulation. The following figure (Fig. 15) represents the different parts that take place in this process.

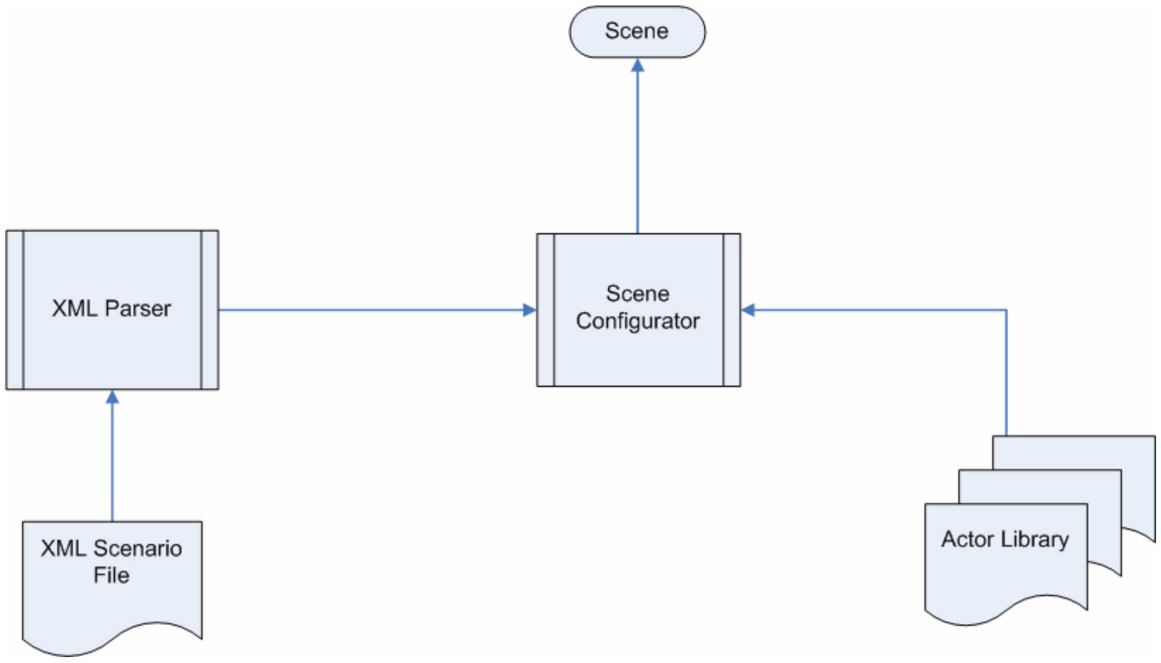

Figure 15. Ship data structure in the Scenario File showing data for example.

When a Scenario File is loaded, the scenario and weather data is read and applied to the Scene, and then the XML parser will read from the file, actor by actor. Each time an actor is detected, the Scene Configurator application will verify that the actor is a valid one using the Game Actor Library; if so, the proper actor will be created, otherwise it will be discarded.

After the actor has been created, the Scene Configurator will create new instances of the Sensor, Communication and Weapons Classes as required for a given actor and will assign them to it

Given that every actor is created inside an instance of the Game Manager, these new actors are capable of receiving and sending game messages and game events, interaction between them is provided by these means and they are controlled and managed by the users, the Game Manager architecture, and the Custom Game Manager Components that will be described in Chapter IX.

Independently from the scene population process, actors can be created or destructed at any time in the simulation, for example a missile, will be created just before being launch, and will be destroyed when it hits its target or get destructed by other means.

Using this approach, different scenarios can be created and loaded into the simulation without code recompilation or application recoding. By separating the data from the implementation of the actors' behaviors, modularity is maintained and allows easier maintenance and improvement.

VII. NETWORK STRUCTURE

A. NETWORK ARCHITECTURE DESCRIPTION

As the name implies, networking is a vital component of a NVE. The network component can be coded separately from other simulation capabilities as a Game Manager Component, thus maximizing modularity. In the current stage of development of the Delta3D game engine, there exist only HLA network connectivity for the Game Manager architecture, but in the future they will be a more game-oriented implementation of network communications.

Meanwhile the proposed architecture uses an implementation of a Client-Server configuration for network communications purposes. The following figure (Fig 16) represents how different stations (computers) will be interconnected in this design.

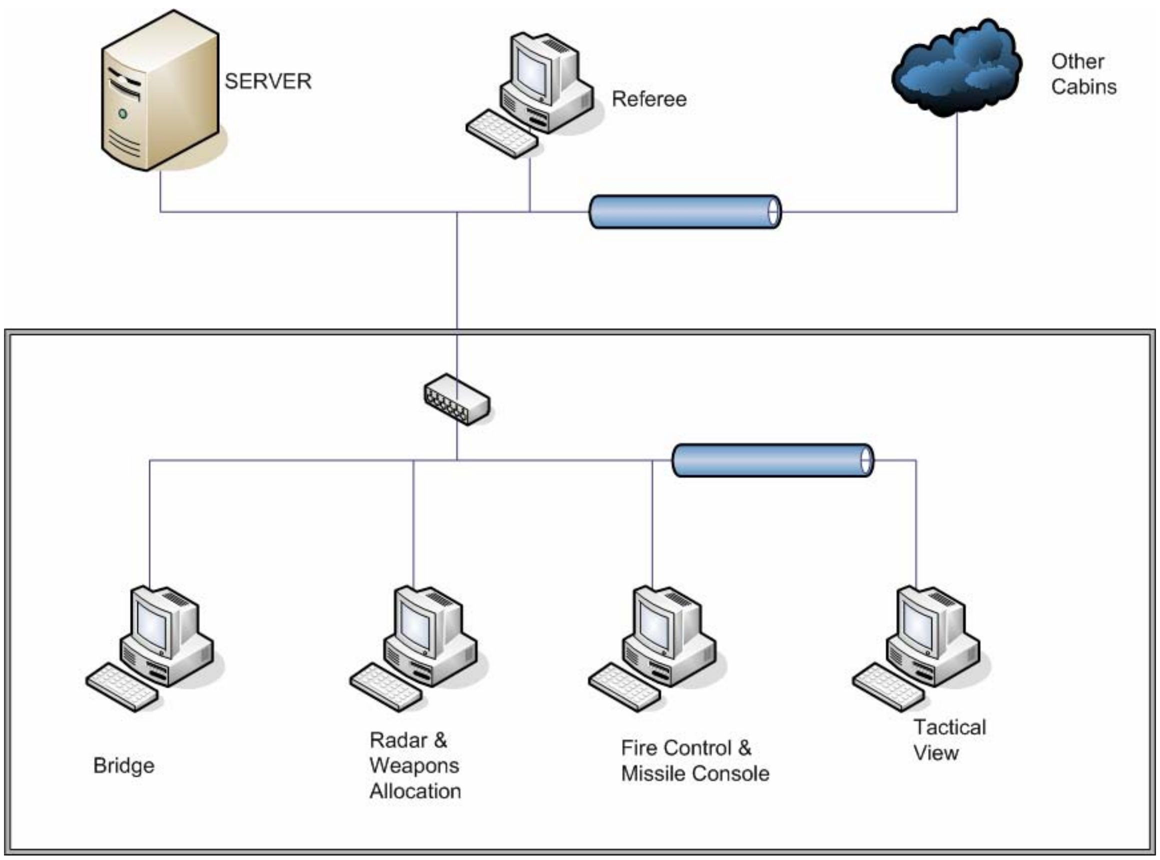

Figure 16. Network configuration showing different stations

A cabin, represented as a gray line, is where different units' stations are being simulated and the outside represents the server side of the network and other units being simulated.

One cabin represents one physical room where several stations can be simulated. As shown in Figure 16, there are four stations inside the cabin. These are the Bridge Station, which will be used as the main node inside a platform, and three other stations (Radar, Fire Control and Tactical View Consoles). While these four computers represent a single unit, they are in communication with the server, as will be seen in Chapter X, this design permits the network to increase the number of nodes in the future while allowing the implementation of certain capabilities not considered in this architecture in first place.

The increase in the number of nodes reflects the increase in nodes inside units, such as new consoles for Electronic Warfare or Communications, and it also considers the increasing in the number of units (in different cabins) that can be attached to the simulation.

The computers for server and arbiters are located in a different physical room. Even when only one referee computer is shown, additional may be attached to the network, permitting multiple arbiters as necessary for grading multiple areas, such as surface tactics, submarines tactics, etc., It also can include some computers for observers.

The server will manage network server, weather conditions, sensor arbiter, and computer controlled forces. While the last of these is not included in the development of this architecture, it is nonetheless considered an integral part. These computer controlled forces can be managed using a Custom Game Manager Component, proposed for this architecture and explained in Chapter IX.

As stated above, even while currently there is not a client-server network component for the Game Manager Architecture in production stage, several efforts are being conducted in the Delta3D user community, thanks to the design of the Game Manager architecture, applying a network component to this design will be transparent and will not require redesign of the architecture proposed in this work.

THIS PAGE INTENTIONALLY LEFT BLANK

VIII. REALISTIC CONSOLE DISPLAYS REPRESENTATION

A. GRAPHICAL USER INTERFACE OVERVIEW

> A graphical user interface (or GUI, often pronounced "gooey"), is a particular case of user interface for interacting with a computer which employs graphical images and widgets in addition to text to represent the information and actions available to the user. Usually the actions are performed through direct manipulation of the graphical elements. [10]

In the case of this simulation, the purpose is to represent real-word equipment interfaces in a computer, this real-world equipment being represented are those that can be found aboard ship, submarines, etc., mainly those found in the CIC of this unit. The representation of realistic controls or movement models for aircrafts is out of the scope of this work.

As an objective, the design and implementation of realistic console interfaces is a main objective. Because these consoles representations will provide the required user interface between the system and the users, this interfaces must be as realistic as possible in order to increase immersion in the simulation and providing at the same time valuable training for those using them.

The Delta3D game engine makes use of the Crazy Eddie’s GUI System (CEGUI) as a dependency. The CEGUI is defined by its author as:

> Crazy Eddie's GUI System is a free library providing windowing and widgets for graphics APIs / engines where such functionality is not natively available, or severely lacking. The library is object orientated, written in C++, and targeted at games developers who should be spending their time creating great games, not building GUI sub-systems! [11].

By using the GUI system inside the Delta3D engine, more realistic consoles can be obtained, which are configurable outside the application code. This GUI’s can be defined by a XML file which provides all data required to create such consoles. The next figure (Fig. 17) shows a possible console representation using the CEGUI.

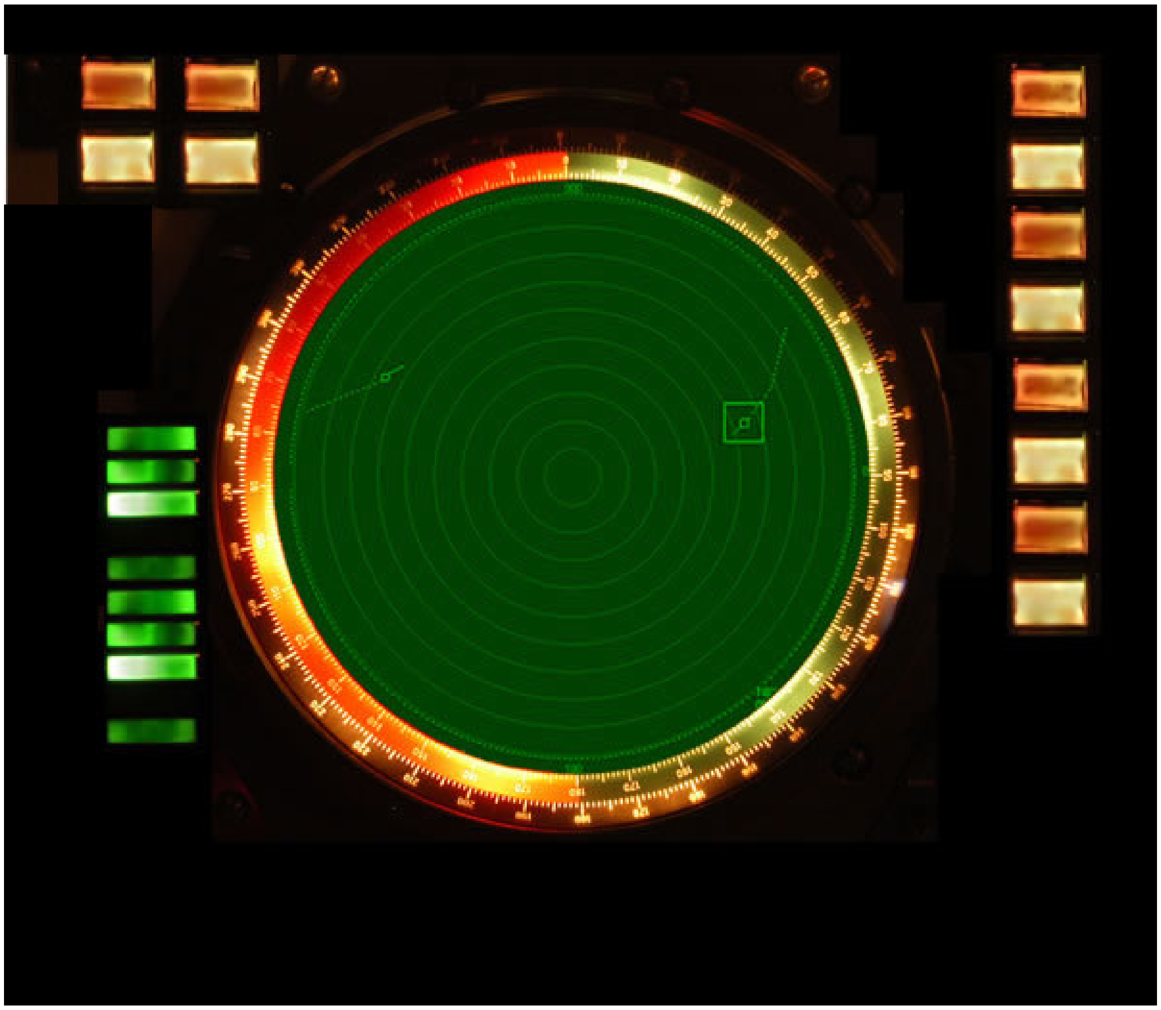

Figure 17. Simulated Radar Console using the GUI capabilities in Delta3D.

The flexibility given by the XML configuration for GUI's allows the creation of different consoles, which can be adapted to represent most real life equipment consoles.

B. GUI CONSTRUCTION IN DELTA3D

In order to construct a GUI inside Delta3D, it is necessary to create three files:

- .layout
- .imageset

- .scheme

The .layout file provides the arrangement of components in the simulation window, by changing this file; different outlooks can be obtained in the GUI.

The .imageset file provides the images required to represent the buttons, scroll bars, and different widgets for the window. The number of image sets that can be used is not limited, which provides capabilities for representing a wide variety of equipment.

The .scheme file provides a means for the application to load all the information previously described.

Generic code to load and represent the GUI is coded inside the application. This code will handle the XML files to load the required images and set the appropriate layout.

To perform event handling more code is necessary. This code can be designed so that common buttons for all consoles can be created, as well as specific ones. Depending on the XML layout that has been loaded, only certain event handler functions are activated, preventing the application from crashing by calling a non-existent GUI widget.

By changing the XML files described above, and without doing any modification in the source code, different consoles and different configurations can be implemented, this will facilitated the creation of new consoles' GUI's for equipment not designed previously and will ease maintenance of the application.

THIS PAGE INTENTIONALLY LEFT BLANK

IX. CUSTOM GAME MANAGER COMPONENTS AND CUSTOM GAME EVENTS

A. GAME MANAGER COMPONENTS OVERVIEW

The Game Manager Architecture makes use of Components as one of the main tools for implementing the architecture. These Components are defined in the software design document as:

> A Component is a special type of object that works with the Game Manager. Components receive all messages and can therefore work at much higher level than an actor… Components perform all sorts of behavior like network connectivity, message logging, and rules validation. Like actors, you can add Components dynamically at either runtime or compile-time. The component is a direct result of the component based design discussion in the Game Actor Layer technical document… [6]

The proposed architecture in this work makes use of several of these Components to facilitate the management of the simulation, as well as a means to provide better modularity by separating several core functions into different elements. This allows future expansion of the simulation.

Every communication inside the Game Manager Architecture is based on messages; these messages are described as follows:

> A Message is a single event within the system, plus all the data that defines that event. [6]

Messages carrying Game Events are extensively used in this simulation. The concept of Custom Game Events is described later in this chapter.

B. CUSTOM GAME MANAGER COMPONENTS OVERVIEW

Given the fact that Components increase modularity by providing encapsulation of functions, methods and data in different sections of the application, this simulation architecture proposes the following Custom Components to manage actors and their behaviors inside the simulation:

- Sensor & Communications Arbiter Component

- Engagement Arbiter Component
- Internal Communication Component
- GUI Component
- Input Component
- Network Component (Client/Server)

The Sensor & Communications Arbiter Component is used to determinate "who sees whom" and "who hears whom" inside the simulation. This component is divided in two sides: client and server.

The Engagement Arbiter Component will resolve the problem of determining weapons impact and damage to the different actors in the simulation. This Component has only a server side.

The Internal Communications Component will provide an interface for communications (using the Network Component) between different consoles belonging to the same unit, for instance, the communication between the console representing the Bridge and the consoles representing the different equipments in the CIC of any given ship as described in Chapter VIII. This Component is also divided in client and server sides.

The GUI Component will implement the Graphical User interface described in Chapter VIII in a Game Manager Component. In this way, all functionality provided by the GUI can be coded separately from the simulation and different kinds of GUI layouts can be created in such a way that they are interchangeable in the application without further modification of the architecture.

Following the same approach, the Input Component will encapsulate all keyboard interfaces handling, providing highly customizable keyboard interfaces for different needs depending on the units being simulated. This Component, as well as the GUI Component, exists in a side-less fashion; given that they can reside in the server or the clients without further modifications other than providing the proper user interface.

Finally the Network Component will provide all the required network communication capabilities to perform the Client-Server connections described in Chapter VII. By coding the network infrastructure as a Component the same benefits of modularity are obtained. This Component is present on both the client and server sides.

It is worth to state that the Network Component is intended as a core Component of the Game Manager Architecture, in conjunction with the Base Input Component, the Dead Reckoning Component, the Default Message Component, the Log Controller Component, the Rules Component, the Server Logger Component, the Task Component and the HLA Component. Several of these components are in developmental stage in the Delta3D project; others, such as the HLA Component, are fully functional in the current version. [12].

As described in Chapter III, Actors and Components generate and response to Game Events, by creating a whole set of Custom Game Events the actors and Components can react to specific simulation events, as well as generate their own events, these Custom Game Events will be described in the following section.

C. CUSTOM GAME EVENT MESSAGES.

Custom Game Events provide the actors and Components of the simulation with a "common language" to communicate actions inside the simulation. Furthermore Custom Game Events provide generalization of events, so each actor or Component can react accordingly to the event, even if these events result in different reactions on different actors and/or Components.

Making use of the Messaging Architecture inside the Game Manager Architecture, these Game Events are sent to all Actors and Components. Their reactions, if any, to these Game Events depend on how they have been initialized.

For instance, the "1/3Ahead" Game Event will only trigger a reaction on Ship and Submarines Actors, but the result of the process triggered by this event will be different quantitatively depending on the Ship or Submarine Type property. In other words, for a Nuclear Carrier Vessel the “1/3Ahead” event will increase the engines’ power to a quantitative higher value than the power required in a small Ossa Class Ship to get the same engine speed.

In other hand, some events like “TargetEngage” will have different qualitative meaning for a ship than for a missile. In a ship, the “TargetEngage” event will enable some weapons to fire, while in a missile the same event will cause the missile to compute and follow a collision path to its target.

The Custom Game Events described in this section are considered as supplementary to the core Game Event Message already described in the Delta3D Game Manager Architecture [6], given that the core Game Events Message are intended for common communication between actors and Components, and the Custom Game Events are specific to this simulation architecture.

D. THE SENSOR & COMMUNICATIONS ARBITER COMPONENT, A DEEPER LOOK

The Sensor & Communications Arbiter Component has the critical task of judging the visibility of actors relative to other actors; this component performs its task in complete independence of the type of actors involved in the detection process, as well as the type and number of sensors involved. This functionality is critical in a naval warfare simulation, given that knowing the position and characteristics of the rival increase greatly the change of survival and victory.

The Sensor & Communications Arbiter Component is divided in two sides: server and client. The main differences between these sides are: where they live in the simulation and what action they perform in their host application (client or server).

In order to clarify the concept of actors and Components communicating via Game Events Message, in this section the Sensor & Communications Arbiter

Component is described using two common examples: target detection by active sensors and target detection by passive sensors.

1. Active Sensor Detection Example

The process required for active detection of targets is described in the following Figure (Fig. 18).

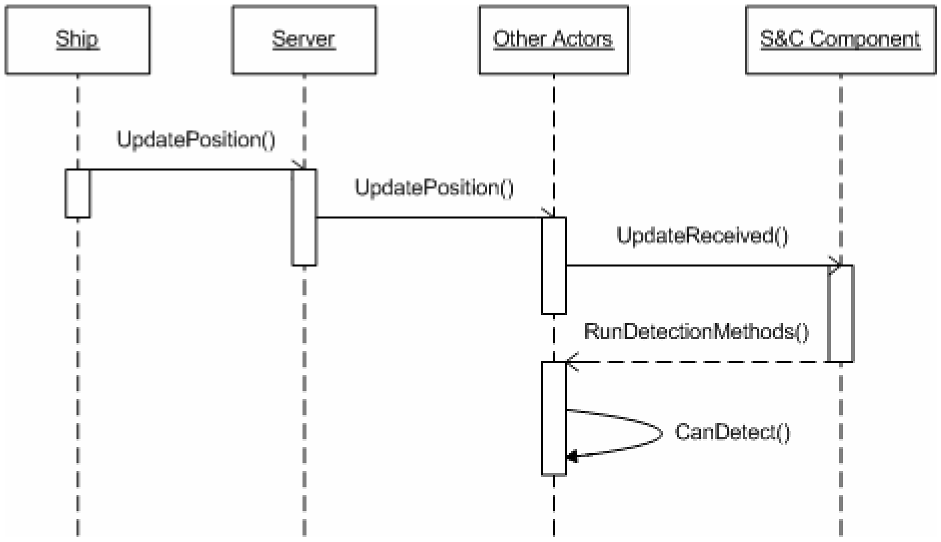

Figure 18. Game Event Message Flow in Actors and Components for active detection.

This process is independent of the kind of units performing the detection, as well as being independent of the type of sensors involved (while being active detection capable and in active mode).

Every time a ship (or any other unit) updates its position in the simulation, it notifies the server its new position by triggering an UpdatePosition() method, which will send an UpdateMessage containing the corresponding updated position (sending the message via the Client Network Component, not shown for simplicity). The server will get the message via the Server Network Component, which will transmit the message to the server side of the Sensor & Communications Arbiter Component. This Component will get the information about the new position of the given unit and will retransmit this information to all other actors and Components in the simulation (except the sender). As Actor receives the message, it will inform their own Client Sensor & Communications Component with the new data received from the server. This component will then fire the detection methods in each sensor that the ship owns; if any of them is capable of detecting the target, the target will be appear as a new target in the unit's consoles or the new position will be updated and displayed if the target was previously in detection range, otherwise nothing happens.

2. Passive Sensor Detection Example

In the case of passive detection, a process very similar to active detection occurs. However, the main difference between these two process are the messages involved and the actions that those messages trigger. The following Figure (Fig. 19) illustrated the process of passive detection.

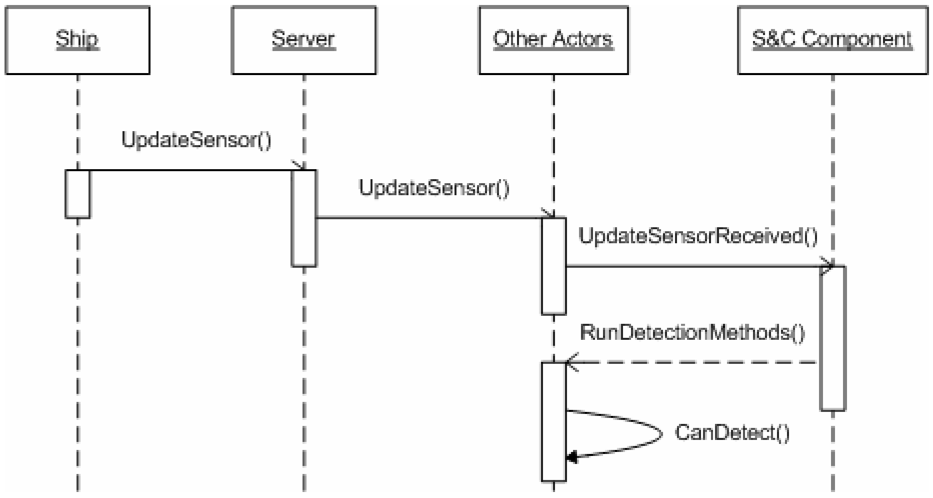

Figure 19. Game Event Message Flow in Actors and Components for passive detection.

In this case, when an actor turns ON its radar (or any other emitting sensor) it sends a UpdateSensor message, following the same message route than active detection, all other actors are informed of such change in the electronic environment and react in consequence.

It can be observed that the methods triggered by the UpdatedSensor Message are different from the methods triggered by the UpdatePosition message. The ability of each actor to react to these types of messages is designed in the very unit class itself. It does not make a difference what kind of ship (or any other unit kind) was initialized by the Scenario File; all instances of the ship class know how to react to such messages, and the same applies for all other actors capable of owning sensors.

THIS PAGE INTENTIONALLY LEFT BLANK

X. INTEGRATION OF THE ARCHITECTURE CONSTITUENTS

A. THE TOTAL IS MORE THAN THE OF SUM OF THE PARTS

The present thesis work has presented an architecture based on five main constituents:

- Game Actor Library
- Scenario Creation, Edition and Configuration
- Network Structure
- Realistic Console Displays Representation
- Custom Game Manager Components

This chapter will explain how the integration of all of them creates the base design to build a Naval Tactical Simulator making use and extending the Game Manager and Dynamic Actor Layer Architectures of the Delta3D game engine.

B. INITIALIZATION OF THE SERVER

The main simulation scene is created in the server system, by reading data from the scenario file and creating and loading actors by generating instances of the proper classes contained in the Dynamic Actor Layer. The simulation server is ready to start the simulation. These steps are carry out as described in Chapter VI, making use of the Dynamic Actor Library describe in Chapter V.

C. INITIALIZATION OF CLIENTS

The proposed initialization is as follows:

- The first computer to connect to the server (a Bridge console application) will choose which unit from those available for human control it will handle.

- Then the next system to connect will choose the proper console that conforms the unit being controlled by the previously mentioned system.
- The previous step is repeated until all consoles are manned.
- Other systems will then connect to control other available human controlled units.
- The simulation is ready to start, activated from the server.

D. CONFORMATION OF THE SERVER AND CLIENT SYSTEMS

The following Figure (Fig. 20) shows how the server and clients systems are conformed by Components.

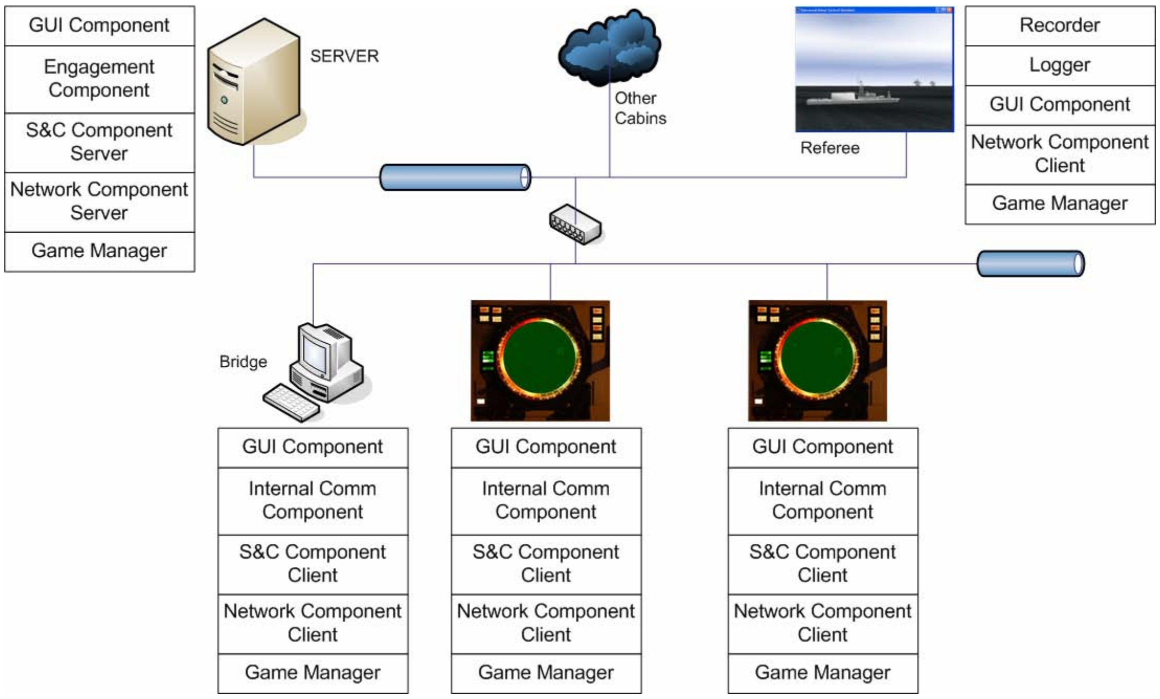

Figure 20. All Architecture constituents working together.

As observed, different components are joined together depending of the system being constructed (server or client).

The server system (in the upper left side of the Figure) is conformed by a Game Manager instance, which makes use of a GUI Component, an Engagement Component, a Server Sensor and Communications Component, and a Server Network Component.

Clients (three of them, in the bottom of the Figure) are conformed by an instance of the Game Manager with the following Components: a GUI Component, an Internal Communication Component, a Client Sensor and Communications Component, and a Client Network Component.

The Game Manager core in the server will be very different from the clients' ones, given that it will contain all necessary functionality for loading scenarios. But in other hand, even when a Bridge Console application will be different from an EMC[7] Console, the core Game Manager clients will not be different at all; the required functionality in each system will be given by the type of Components loaded on start up.

This will allow any client application to be run as a Bridge or any other CIC console by loading the corresponding Components, which can be assigned in the initialization XML file of each system, or chosen from a menu on star up. It will be required that all necessary Components are present on client systems in order to be capable of such initialization.

E. CUSTOM "OBSERVER" SYSTEMS

In the upper right of Figure 20, there is a system that does not perform any of the mentioned tasks as a server neither a client system in the simulation.

This system is conformed by an instance of the Game Manager and the following Components: a Recorder Component, a Logger Component, a GUI Component, and a Client Network Component.

This system is capable of observing the simulation by connecting to the network and getting all messages being sent, it is also capable of recording and

[7] Electronic Counter Measures

logging every event in the simulation. The Game Manager in this system is the same Game Manager core used in client systems.

Individualization of systems can be carry out by attaching different Components to core Client Game Managers.

By using this architecture, different components that do not depend upon each other can exist in different computers. For instance, there is no impediment to the Logger Component (shown in the Referee system) to be located in a different computer than the Recorder Component.

By separating components and hence simulation tasks in different computer systems, these computer systems can run with less load, and the total load of the simulation is divided.

XI. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

The present thesis work made extend use of the new Delta3D capabilities given by the Game Manager and Dynamic Actor Layer.

By extending these architectures, it was possible to create a new one, which allows passing from the generalized game structure given by Delta3D to a specialized one for a naval tactical simulation, which follows the principles of object oriented programming and modularity.

Ships, submarines, aircrafts, weapons, sensors and communications are designed using a generalized approach that allows individualization of them by the data provide by the user through the scenario file.

The use of data from different sources, public or classified, is possible by changing de data required to build the scenario already described. No further chances in the simulation are required for that matter.

Mentioned scenarios are capable of being easily designed, created and edited, the XML technology made this possible by providing a Meta language specifically designed to solve this issue.

B. POSSIBLE FUTURE WORK

The author of this thesis expects to continue enriching the proposed architecture by building a working application based on it in his next assignment.

There exist several possible enhancements to architecture described in here, such as:

Create an AI module, this module can be used to control Computer Generated Forces for friendly, enemy and neutral units, as described in the body if this thesis, such module can be designed and implemented without major modifications to the base architecture if it is built as a Custom Game Component.

As described in the Limitations Section of this work, this design does not included the development of sensors, weapons and units’ movement models, this area can be improved by building the proper supporting classes which can be incorporated to the architecture as described in Chapter V, improvement on the supporting classes is already considered as a feature of this architecture and will not alter its overall design.

By using Custom Game Components several functions as After Action Review, Simulation Events Logging, Record and Playback functions can be incorporated, these Game Components can be included in the server system or they can be installed in a specifically designated computer system, such that the total system load is balanced over all computer in the simulation. The general description of this process is established in Chapter X.

Finally there exist several enhancements that could be implemented to improve this architecture, some of them as terrain rendering or area of interest management will require modification of the architecture in some level given that the core of the simulation, the Server Game Manager, needs to be redesigned to incorporate such features; while some other features as dead reckoning algorithms can be designed as a Custom Component or a supporting class which will not affect the architecture. The fields for improvement are left open for future users of the Delta3D game engine to pursue them.

LIST OF REFERENCES

1 Gibson, Dick, “Some interesting oil industry statistics”, in Gibson Consulting wbesite, http://www.gravmag.com/oil.html (accessed September 10, 2006)

2 Simmons, Matthew R., “The World’s Giants Oilfields”, in Simmons and Company International Website, (9 January 2002 [cited 10 September 2006]); available form the World Wide Web @ http://www.simmonsco-intl.com/files/giantoilfields.pdf

3 Martinez Tiburcio, Felix, “Maritime Protection of Critical Infrastructure Assets in the Campeche Sound”, (Master’s Thesis, Naval Postgraduate School, 2005)

4 Wikipedia contributors, "Game engine," *Wikipedia, The Free Encyclopedia,* http://en.wikipedia.org/w/index.php?title=Game_engine&oldid=74829032 (accessed September 10, 2006).

5 BMH Associates, Inc, “Dynamic Actor Layer (DAL)”, BMH Associates Inc. (29 August 2005 [cited 10 September 2006]); available from the World Wide Web @ http://www.delta3d.org/filemgmt_data/files/Delta3D_DAL_SDD_1.2.pdf

6 BMH Associates, Inc, “Game Manager”, BMH Associates Inc. (26 October 2005 [cited 10 September 2006]); available from the World Wide Web @ http://www.delta3d.org/filemgmt_data/files/Delta3D_GameManager_SDD_1.0.pdf

7 Wikipedia contributors, "Modular design," *Wikipedia, The Free Encyclopedia,* http://en.wikipedia.org/w/index.php?title=Modular_design&oldid=58988694 (accessed September 10, 2006).

8 Wikipedia contributors, "Object-oriented programming," *Wikipedia, The Free Encyclopedia,* http://en.wikipedia.org/w/index.php?title=Object-oriented_programming&oldid=74684082 (accessed September 10, 2006).

9 Wikipedia contributors, "Extensible Markup Language," *Wikipedia, The Free Encyclopedia,* http://en.wikipedia.org/w/index.php?title=Extensible_Markup_Language&oldid=74592740 (accessed September 10, 2006).

10 Wikipedia contributors, "Graphical user interface," *Wikipedia, The Free Encyclopedia,*

http://en.wikipedia.org/w/index.php?title=Graphical_user_interface&oldid=73834154 (accessed September 10, 2006).

11 Crazy Eddie's GUI System, “Main Page”, Crazy Eddie's GUI System Website, http://www.cegui.org.uk/wiki/index.php/Main_Page (accessed September 10, 2006).

12 Delta3D, “Features”, Delta3D Open Source Gaming & Simulation Engine, http://www.delta3d.org/article.php?story=20051209133127695&topic=docs (accessed September 10, 2006).

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. General Staff of the Mexican Navy
   Mexican Navy Headquarters
   Mexico Distrito Federal
   Mexico

4. Centro de Estudios Superiores Navales
   Mexico Distrito Federal
   Mexico

5. Perry L. McDowell
   Naval Postgraduate School
   Monterey, California

6. Dr. Rudolph Darken
   Naval Postgraduate School
   Monterey, California